Data mining di Dati Scientifici

Manuel Jerez González

Codice Persona: 10844889

Advisor: Prof. Barbara Pernici

—

Project in Computer Science and Engineering
Academic Year 2021/2022

—

Politecnico Milano

Index

1. Introduction

The aim of this report is to explain the problem, context and solutions proposed during my work for the 2021/2022 edition of the course “Project in Computer Science and Engineering” from Politecnico Milano. I decided to take the project named “Data mining di dati scientifici” tutored by Professor Barbara Pernici.

The work done is framed in the SciExpeM project, powered by the Electronics, Information and Bioengineering Department (DEIB) and the Chemical, Materials and Chemical Engineering Department (CMIC) from Politecnico Milano.

2. Context

The goal of the project SciExpeM is to support the development and maintenance of predictive models on combustion kinetics, that help in the future to improve the efficiency, increase sustainability, reduce chemical pollutants, and help in the research and development of green fuels. This is done thanks to the increasing amount of experimental data available on this topic, and also thanks to the high computational resources on nowadays PCs that allow the project to use machine learning and other general data science techniques to push forward on the challenges of extracting data, developing models, simulating the combustion processes, and analysing the results.

The studies on the combustion kinetics have been carried on and its data has been collected since the end of the WWII, and they continue today. Of course, the methods for acquiring the experimental data and the form of representation have been changing along the time, which leads to some uncertainties and the necessity to work on the data. This experimental data will be checked against the models and simulations developed by the project, and the results of the comparison will be analysed, in order to develop and enhance the performance of the model. This means that the project manages three types of data: Experiments, simulations and models.

For my topic of research, the focus is on the kinetic models, which describe the different elements that appear in the chemical reactions that happen during the combustion. These kinetic models contain information on the species and reactions physical properties, which are needed to execute the simulations carried on by the project. The extraction of data from models is crucial, as we need to understand which model is better between some of them, as well as identify similarities and differences between the components present on them.

Having said that, it is worth explaining how we take the data into our software. All the data needed to specify a chemical kinetic model is contained in two different XML files, which both have a fixed schema. The first file contains the number of reactions in the model, as well as all the reaction names present on it. The other file, much more complex, contains the metadata of the model, the elements and species that are considered and all the chemical information and properties for the species and reactions.

The approach that we followed to organize my work was to have meetings with my tutor every 2 weeks on average, discuss the work previously done and set the tasks that I had to do for the upcoming weeks.

3. Problem

As it has been said, to achieve the latest goal of SciExpeM, we need to be able to understand the data we are using, as there is a huge number of different ways to express the information we need to manage. One of the things we need to observe is the different possible kinetics models that are available where the chemical components are defined and inserted into the system.

The work that I was requested to do follows this topic, which is, given different chemical kinetics models, whose formats will be explained later, be able to compare them, disambiguating the different elements of the files with all the data and, with this information, conclude if species and reactions given are the same even if the chemical models and names used are different.

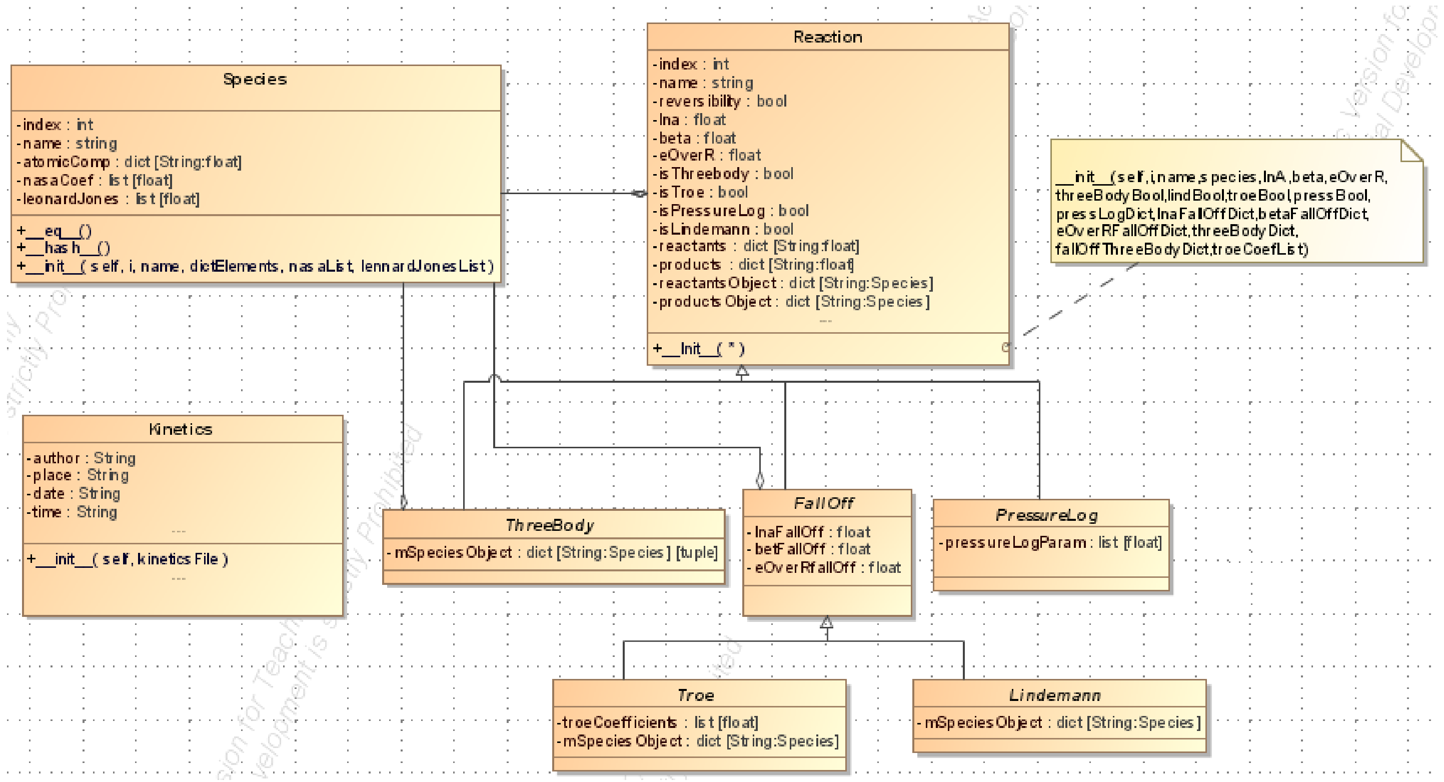
To do this, nevertheless, we need to attend other topics, such as the acquisition of data from specific files, the creation of models on Python to store the information and stablish ways to compare two species coming from different kinetic models.

4. Solutions

4.1. Understanding data and creating Python models

The first thing that was to be done to start working on the project was to understand the type of data that we had on the files, so we could decide how many classes, and which attributes had to be created. The first step we did for this purpose was creating the XML schema for both input files, so we could easily see all the properties that were to be developed and included in the Python classes.

Then, once this was understood, we could start working on the model. Before coding, I decided to make an UML diagram, to reflect the structure of the model, which is detailed below.



Species
-index : int
-name : string
-atomicComp : dict [String:float]
-nasaCoef : list [float]
-leonardJones : list [float]
+__eq__()
+__hash__()
+__init__(self, i, name, dictElements, nasaList, lennardJonesList)
Reaction
-index : int
-name : string
-reversibility : bool
-lna : float
-beta : float
-eOverR : float
-isThreebody : bool
-isTroe : bool
-isPressureLog : bool
-isLindemann : bool
-reactants : dict [String:float]
-products : dict [String:float]
-reactantsObject : dict [String:Species]
-productsObject : dict [String:Species]
+__Init__(*)
__init__(self,i,name,species,lnA,beta,eOverR, threeBodyBool,lindBool,troeBool,pressBool, pressLogDict,lnaFallOffDict,betaFallOffDict, eOverRFallOffDict,threeBodyDict, fallOffThreeBodyDict,troeCoefList)
Kinetics
-author : String
-place : String
-date : String
-time : String
+__init__(self, kineticsFile)
ThreeBody
- mSpeciesObject : dict [String:Species] [tuple]
FallOff
- lnaFallOff : float
- betFallOff : float
- eOverRfallOff : float
PressureLog
-pressureLogParam : list [float]
Troe
-troeCoefficients : list [float]
-mSpeciesObject : dict [String:Species]
Lindemann
- mSpeciesObject : dict [String:Species]

Figure 1. Class diagram.

The coding language we used for our project is Python. The reasons behind this decision are that this language suits perfectly for managing huge amounts of data, as it has different libraries to easily extract from files and work with it. Apart from that, it gives us flexibility with types and structures and allows us to create classes and work with them.
There are some things to comment about the diagram. The first, the __*init*__ method, which is the constructor for Python objects. In the case of the *Species* class, we also have the __*eq*__ method, which is used to compare the atomic composition of two different, in order to use them in conditions operators; the __*hash*__ function is needed to use the class *Species* as the key value for dictionaries.

The complexity of the diagram and the model itself comes when talking about the reactions, as the data that has to be stored there depends on some values, because the reactions can have different types attending to physical attributes. In our model, the class modelling the reactions is unique, thanks to the Python flexibility when creating objects and its attributes, but on the diagram it is represented with different classes. As an example, all reactions will be an instance for the Reaction class, but if the attribute *isThreeBody=True*, the object will have the properties of the ThreeBody class. Same will happen with TROE, Lindemann and Pressure Log reactions.

Another thing to point out is that in the case of the ThreeBody, TROE and Lindemann reactions, the attributes *reactants* and *products* will have a special key called *M* and its associated value will be a dictionary containing pairs *[String:Float]* representing the possible values for *M*.

There are two properties on the definition of the species in the XML files which are *Stoichiometry* and *Transport* that I was told not to model as they are not needed at the level we're working with. Also, for the first of them, the project has already developed a Python class to access this property.

4.2. Reading data

As explained in the "Context" section, the info that we need to read is contained in two XML files, that follow a schema predefined.

The first thing we will attend is the insertion of the species, as it is easier than the reactions. This subject is done by the function *getSpecies* (on the *getSpecies.py* file) and the auxiliar function *parseSpecies* (on the *parseSpecies.py* file), where the second one is called by the first one.

The *parseSpecies* function gets as parameter the file that must be read (the one that defines the kinetics of the model), and using the xml Python library, it extracts the elements that characterise the species. All this information is returned into lists to the *getSpecies* function, and then iterates to create one Species instance for each of the elements that were read. After the creation of all the instances of the *Species* class, then the *Kinetics* object is created. The *getSpecies* function, which is called by the application to be able to interact with the model, returns a list of *Species* objects and the *Kinetics* object.

This first part of the insertion of data in the system was quite straightforward, as all the species have the same type of properties, so the challenge was to read them correctly and insert them into the different objects.

Now, we move on to read the files and get from them the information of the chemical reactions that are given in the model. On a parallel way as it happened with the species, we do that with *getReactions* function (contained on the file getReactions.py) and the auxiliar *parseReactions* function (on the parseReactions.py file).

The *parseReactions* function gets two parameters, which are the names of the two files that define the model: the kinetics file and the file containing all the names of all the chemical reactions. Again, this function will use the xml library and return several lists. The *getReactions* function gets three parameters. The first one is the list of the species that have been read from the model. Then, the names of the files that are needed to get the information. On this function the first action is to call the *parseReaction* function, and then create a list of chemical reactions object, which is returned to the user.

This part of the process is much more complex, as depending on the type of chemical reaction, the object will have different properties, which some of them require to access the species list. This means that the parsing of the file takes more time and returns more objects; and the creation of each single *Reaction* object needs to check the different types of the reaction, in order to include or not more properties than the commons to all the different reactions.

In this diagram we can see the sequence of calling all the functions explained above, that are needed to get the Python objects that represent the whole model. This diagram does not suit perfectly to the UML standard, but it is used as an auxiliar way to clarify the interactions on this part of the software.

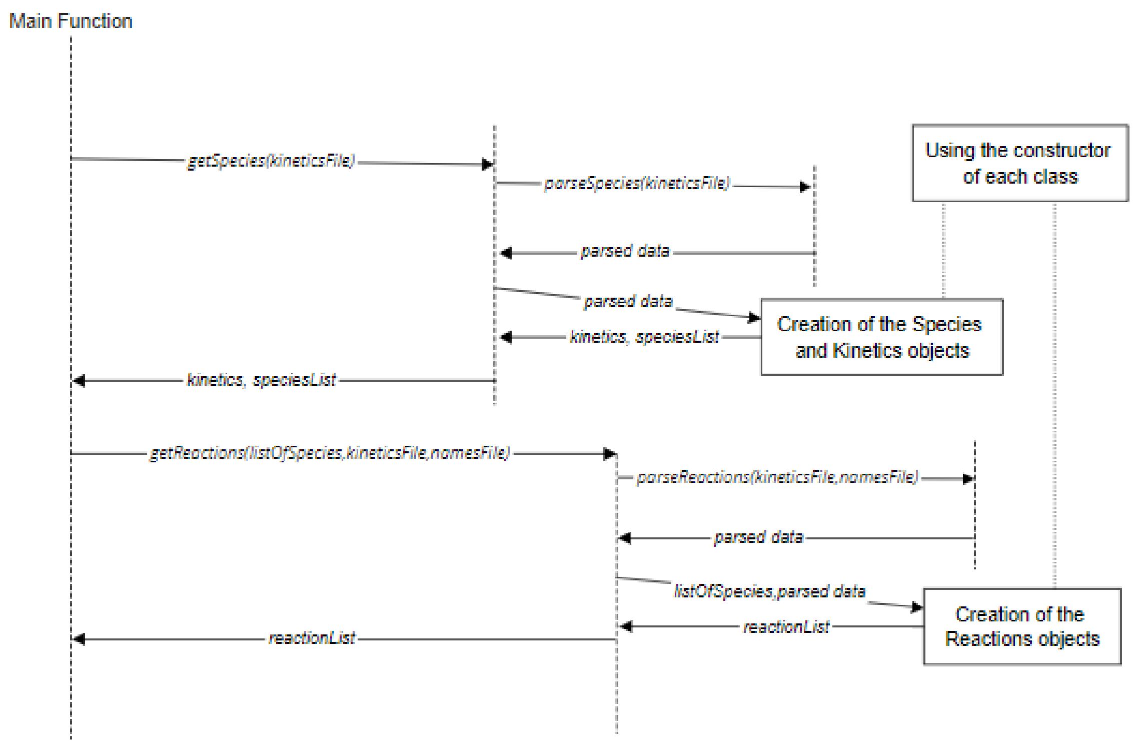

Main Function
getSpecies(kineticsFile)
parseSpecies(kineticsFile)
Using the constructor of each class
parsed data
parsed data
Creation of the Species and Kinetics objects
kinetics, speciesList
kinetics, speciesList
getReactions(listOfSpecies,kineticsFile,namesFile)
parseReactions(kineticsFile,namesFile)
parsed data
listOfSpecies,parsed data
Creation of the Reactions objects
reactionList
reactionList

Figure 2. Sequence diagram.

4.3. Plotting species networks

The next step in the development of the project was the creation and visualization of the species network. The idea was to create an undirected node graph, where the nodes would be the species and there would be connections between them if they appeared in the same chemical reaction.

For this purpose, I used two different external libraries. The first one is *NetworkX*, a library that gives us methods to easily create nodes and edges, as well as an easy way to access properties of the network. The other library used is *plotly*, which provides us a way to plot graphs (a network graph in our case) which will open in the web navigator. This allows us to interact with the graph, what is really important in our case, and is a great advantage in relation to other libraries more frequently used like matplotlib.

All the methods related to the plotting of the network (and other tasks related to the network disambiguation) are contained in the file called *networkFunctions.py*. The user will call the method *getNetwork*, which gets as parameters the species list, the reaction list, and a list of the names of species which want to be seen on the graph. The reason behind the last parameter is that the model that I was given to train had more than 500 species and 20000 reactions, so plotting the whole network graph was too time consuming due to the complexity. Because of that, the user will specify the species in which he is interested, and the network plotted will contain only the species with relations to those.

The already explained function uses two auxiliar methods, which are *generateNetwork* and *plotNetwork*. The first one gets the species and reactions lists and returns the network object, while the second one gets the species that the user wants to see and the network object and plots the result, opening it on the default web browser.

4.4. Comparing species

As it was already explained before, the goal of the project was to be able to compare species, to be able to disambiguate them, telling if they are referring to the same component even if they have other names or the properties are differently expressed. The core of this functionality is on the file compare.py, but it uses other methods, as will be explained later.

The method to start the comparison between species is *compareSpecies*(contained in *compare.py*). It receives as parameters the name of the species from one model which wants to be compared, a list of the names of species against it will be compared with, two lists of species and two lists of reactions (one of each for each of the models being compared).

This function will use the *generateNetwork* function, which was explained before, and the *getDistanceBetweenSpecies* method (contained also in *compare.py*), which gets as parameters two species and its two network graphs, in order to get different values of distance, related to the physical/chemical properties and the network distance.

All the values received from the different comparisons between species will be added to different arrays and plotted into bar graphs, using the library matplotlib to display the graph.

At this point, it is worth explaining which is the method that we used to determine the difference. We use three values, which will be considered as "coordinates". The first one is the NASA-Coefficient distance. It is computed as the square root of the sum of difference between NASA-Coefficients values (ordered, so the lowest value in one species is compared with the lowest on the other) to the power of two. The same approach is used with the Lennard Jones coefficients, which is our second coordinate. Then these two values are normalized, dividing them into the square root of the sum of the values raised to the power of two. The resulting values will be the ones returned and plotted on the graph.

The last coordinate of the distance is the value measured on the graph. To compute this, we use a function called *getPropertiesOfNode* (contained on *networkFunctions.py* file), which gets as parameters the graph object and the name of the species. This method returns for the species the centrality, betweenness, load centrality, closeness centrality and eigenvector centrality. With these values for both species, we get the square root of the sum of the power of the differences between each of the properties; which is our third coordinate.

Once we have the three values that we use as coordinates for each of the species, we compute the absolute distance, as if the two species were points in a three-dimensional space; with the norm of the vector that would join them.

The result of the function is a bars graph, showing for each comparison between the first species specified and the ones that we wanted to match 4 values, which are the NASA-coefficient distance, the Lennard Jones distance, the graph distance, and the absolute value of the distance. This graph is plotted using the library *MatPlotLib*.

4.5. Ontology of the model

As the last task, the challenge was to implement a function to create an ontology description for the kinetic model. The model follows the *OntoKin* model, developed on this paper. Although there are libraries for Python to develop ontology models, I decided not to use them as the relations on the diagram could be easily accessed with the properties of the species and reactions classes that I already had developed on previous steps of the project.

More concretely, I was requested to create a function that for the kinetic model, this is, the list of reactions and the list of species on the XML files, would create a CSV file with three columns, with the relations described on the paper mentioned before. To do this, I created a function called *ontology*, on the file *ontology.py*, which first iterates through the reaction list, extracting for all the reactions the properties and writing them into different lists. Then, there's an iteration through all the species, to get their element composition, which is the only property that we need to include on the ontology diagram following the *OntoKin* descriptions.

Along the "for" loops through both lists, other lists are created containing the info that is needed to be included on the csv file that is generated at the end of the function. The structure of this lists is the same for all, as they have three columns, indicating the origin node, the name of the relation and the end node. Once all these lists are created, they are stored into the file, using the csv library for python.

One thing that must be noted is that not all the relations and entities described on the *OntoKin* definition are implemented on my function. The reason is that to create all the elements on the model I would need more information than what is contained on the kinetic models that I was given, and that are interesting for SciExpeM project. The entities not implemented, as well as their input and output links, are specified on the code of the function. This file also contains a detailed explanation of all the relations that were implemented, as well as examples of most of them.

The output for this function is a CSV file called *knowledge.csv* in the folder *ontology*.

4.6. Other functions implemented

Aside from the functions that were specified before and aimed to work on the topic of the entity disambiguation, I was also requested to do an independent task.

This task was to create a function that was able to plot the evolution of the forward reaction rate of the different chemical reactions, depending on temperature and pressure. This is done with mathematical formulas already predefined, and my job was to implement them in Python. These formulas are different depending on the type of the reaction, so we would need to create different functions for the Elementary, the Threebody, the TROE, the Lindemann and the Pressure Log reactions. For some of them, the dependence is only on the temperature of the system, and others depended also on the pressure.

Although, as already said, there are different formulas for each type of reactions, I was only requested to do the corresponding to the Elementary and Threebody reactions (it is the same formula for both types) which is only dependant on the temperature. This is implemented in the function *plotKElementary*, which receives as parameters the reaction, the minimum and maximum temperature and plots a logarithmic graph where the forward reaction rate is drawn against the inverse of the temperature.

The formula for these two types of reactions is the following:

$$K(T)=A*T^{b}*e^{-E/(RT)}$$

Where A is the lnA value, b is the Beta value and E/R is the eOverR value, all of them extracted from the kinetics file and included in the objects representing the chemical reactions.

4.7. File structure and usage diagrams

The file structure for the scripts folder is detailed here, as a clarification of the functions and classed implemented and where they can be found.

Apart from the files and functions detailed on this table, I decided to include 4 Python scripts, which can be easily executed in order to try the different functionalities implemented and explained before. These scripts will make use of all the functions and classes implemented below, and the requirements to use them are included on the scripts themselves. Also, in the folder *xml* there is an example of the XML files and its corresponding XML schema, and these are the objects that I used as a reference for developing all the project. Lastly, the CSV file that is used to describe the knowledge graph can be found on the ontology folder, but there is a script that will regenerate it if needed. The previously referred scripts are all named starting with the word “test”. The link to the GitHub repository with all the files and executables is included on the *README.txt* file.

compare.py	findSpeciesByName.py	getReactions.py	getSpecies.py
getDistanceBetweenSpecies	findSpeciesByName	getReactions	getSpecies
compareSpecies			
kinetics.py	**networkFunctions.py**	**ontology.py**	**parseReactions.py**
Class Kinetics	plotNetwork	ontology	parseReactions
	generateNetwork		
	getNetwork		
	getPropertiesOfNode		
parseSpecies.py	**plotK.py**	**reactions.py**	**species.py**
parseSpecies	plotKElementary	Class Reaction	Class Species

Figure 3. Functions structure.

The following graph reflects the functions implemented on the project, as well as the relations between them. Note that the scripts folder also contains four testing scripts, to try the *getNetwork*, *compareSpecies*, *ontology* and *plotKElementary* functions. These files are not included on the diagram below as are not necessary for the general functioning of the project, they are just created as prototypes for showing the different outputs of the functions.

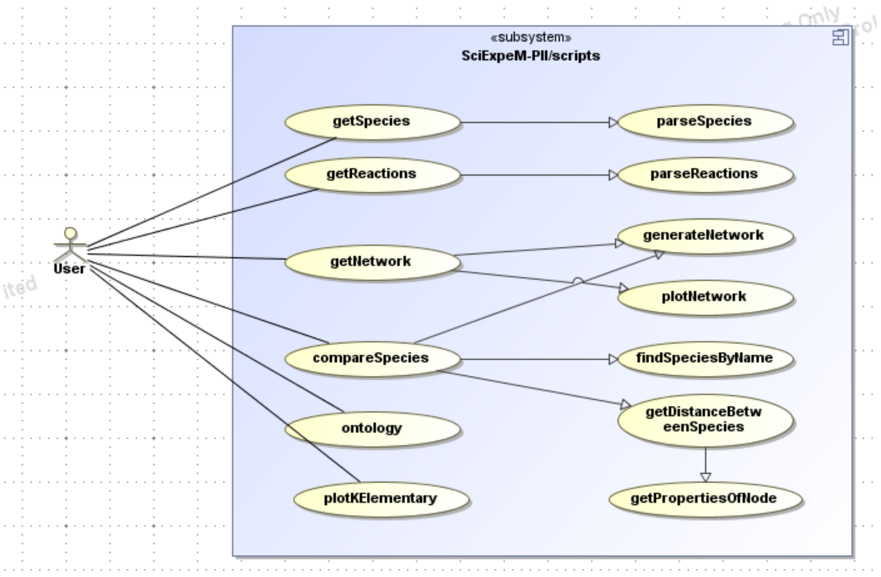


Figure 4. Usage diagram.